NAME

namespace — create and manipulate contexts for commands and variables

SYNOPSIS

namespace *subcommand* ?*arg ...*?

DESCRIPTION

The **namespace** command lets you create, access, and destroy separate contexts for commands and variables. See the section WHAT IS A NAMESPACE? below for a brief overview of namespaces. The legal values of *subcommand* are listed below. Note that you can abbreviate the *subcommand*s.

namespace children ?*namespace*? ?*pattern*?
: Returns a list of all child namespaces that belong to the namespace *namespace*. If *namespace* is not specified, then the children are returned for the current namespace. This command returns fully-qualified names, which start with a double colon (**::**). If the optional *pattern* is given, then this command returns only the names that match the glob-style pattern. The actual pattern used is determined as follows: a pattern that starts with double colon (**::**) is used directly, otherwise the namespace *namespace* (or the fully-qualified name of the current namespace) is prepended onto the pattern.

namespace code *script*
: Captures the current namespace context for later execution of the script *script*. It returns a new script in which *script* has been wrapped in a **namespace inscope** command. The new script has two important properties. First, it can be evaluated in any namespace and will cause *script* to be evaluated in the current namespace (the one where the **namespace code** command was invoked). Second, additional arguments can be appended to the resulting script and they will be passed to *script* as additional arguments. For example, suppose the command **set script [namespace code {foo bar}]** is invoked in namespace **::a::b**. Then **eval $script [list x y]** can be executed in any namespace (assuming the value of **script** has been passed in properly) and will have the same effect as the command **::namespace eval ::a::b {foo bar x y}**. This command is needed because extensions like Tk normally execute callback scripts in the global namespace. A scoped command captures a command together with its namespace context in a way that allows it to be executed properly later. See the section SCOPED SCRIPTS for some examples of how this is used to create callback scripts.

namespace current
: Returns the fully-qualified name for the current namespace. The actual name of the global namespace is “” (i.e., an empty string), but this command returns **::** for the global namespace as a convenience to programmers.

namespace delete ?*namespace namespace ...*?
: Each namespace *namespace* is deleted and all variables, procedures, and child namespaces contained in the namespace are deleted. If a procedure is currently executing inside the namespace, the namespace will be kept alive until the procedure returns; however, the namespace is marked to prevent other code from looking it up by name. If a namespace does not exist, this command returns an error. If no namespace names are given, this command does nothing.

namespace ensemble *subcommand* ?*arg ...*?
: Creates and manipulates a command that is formed out of an ensemble of subcommands. See the section ENSEMBLES below for further details.

namespace eval *namespace arg* ?*arg ...*?
: Activates a namespace called *namespace* and evaluates some code in that context. If the namespace does not already exist, it is created. If more than one *arg* argument is specified, the arguments are concatenated together with a space between each one in the same fashion as the eval command, and the result is evaluated.

  If *namespace* has leading namespace qualifiers and any leading namespaces do not exist, they are automatically created.

namespace exists *namespace*
: Returns **1** if *namespace* is a valid namespace in the current context, returns **0** otherwise.

namespace export ?**-clear**? ?*pattern pattern ...*?
: Specifies which commands are exported from a namespace. The exported commands are those that can be later imported into another namespace using a **namespace import** command. Both commands defined in a namespace and commands the namespace has previously imported can be exported by a namespace. The commands do not have to be defined at the time the **namespace export** command is executed. Each *pattern* may contain glob-style special characters, but it may not include any namespace qualifiers. That is, the pattern can only specify commands in the current (exporting) namespace. Each *pattern* is appended onto the namespace's list of export patterns. If the **-clear** flag is given, the namespace's export pattern list is reset to empty before any *pattern* arguments are appended. If no *pattern*s are given and the **-clear** flag is not

given, this command returns the namespace's current export list.

namespace forget ?*pattern pattern ...*?
Removes previously imported commands from a namespace. Each *pattern* is a simple or qualified name such as **x**, **foo::x** or **a::b::p***. Qualified names contain double colons (**::**) and qualify a name with the name of one or more namespaces. Each "qualified pattern" is qualified with the name of an exporting namespace and may have glob-style special characters in the command name at the end of the qualified name. Glob characters may not appear in a namespace name. For each "simple pattern" this command deletes the matching commands of the current namespace that were imported from a different namespace. For "qualified patterns", this command first finds the matching exported commands. It then checks whether any of those commands were previously imported by the current namespace. If so, this command deletes the corresponding imported commands. In effect, this undoes the action of a **namespace import** command.

namespace import ?**-force**? ?*pattern pattern ...*?
Imports commands into a namespace, or queries the set of imported commands in a namespace. When no arguments are present, **namespace import** returns the list of commands in the current namespace that have been imported from other namespaces. The commands in the returned list are in the format of simple names, with no namespace qualifiers at all. This format is suitable for composition with **namespace forget** (see EXAMPLES below).

When *pattern* arguments are present, each *pattern* is a qualified name like **foo::x** or **a::p***. That is, it includes the name of an exporting namespace and may have glob-style special characters in the command name at the end of the qualified name. Glob characters may not appear in a namespace name. When the namespace name is not fully qualified (i.e., does not start with a namespace separator) it is resolved as a namespace name in the way described in the NAME RESOLUTION section; it is an error if no namespace with that name can be found.

All the commands that match a *pattern* string and which are currently exported from their namespace are added to the current namespace. This is done by creating a new command in the current namespace that points to the exported command in its original namespace; when the new imported command is called, it invokes the exported command. This command normally returns an error if an imported command conflicts with an existing command. However, if the **-force** option is given, imported commands will silently replace existing commands. The **namespace import** command has snapshot semantics: that is, only requested commands that are currently defined in the exporting namespace are imported. In other words, you can import only the commands that are in a namespace at the time when the **namespace import** command is executed. If another command is defined and exported in this namespace later on, it will not be imported.

namespace inscope *namespace script* ?*arg ...*?
Executes a script in the context of the specified *namespace*. This command is not expected to be used directly by programmers; calls to it are generated implicitly when applications use **namespace code** commands to create callback scripts that the applications then register with, e.g., Tk widgets. The **namespace inscope** command is much like the **namespace eval** command except that the *namespace* must already exist, and **namespace inscope** appends additional *arg*s as proper list elements.

```
namespace inscope ::foo $script $x $y $z
```

is equivalent to

```
namespace eval ::foo [concat $script [list $x $y $z]]
```

thus additional arguments will not undergo a second round of substitution, as is the case with **namespace eval**.

namespace origin *command*
Returns the fully-qualified name of the original command to which the imported command *command* refers. When a command is imported into a namespace, a new command is created in that namespace that points to the actual command in the exporting namespace. If a command is imported into a sequence of namespaces *a, b,...,n* where each successive namespace just imports the command from the previous namespace, this command returns the fully-qualified name of the original command in the first namespace, *a*. If *command* does not refer to an imported command, the command's own fully-qualified name is returned.

namespace parent ?*namespace*?
Returns the fully-qualified name of the parent namespace for namespace *namespace*. If *namespace* is not specified, the fully-qualified name of the current namespace's parent is returned.

namespace path ?*namespaceList*?
Returns the command resolution path of the current namespace. If *namespaceList* is specified as a list of named namespaces, the current namespace's command resolution path is set to those namespaces and returns the empty list. The default command resolution path is always empty. See the section NAME RESOLUTION below for an explanation of the rules regarding name resolution.

namespace qualifiers *string*

Returns any leading namespace qualifiers for *string*. Qualifiers are namespace names separated by double colons (**::**). For the *string* **::foo::bar::x**, this command returns **::foo::bar**, and for **::** it returns an empty string. This command is the complement of the **namespace tail** command. Note that it does not check whether the namespace names are, in fact, the names of currently defined namespaces.

namespace tail *string*

Returns the simple name at the end of a qualified string. Qualifiers are namespace names separated by double colons (**::**). For the *string* **::foo::bar::x**, this command returns **x**, and for **::** it returns an empty string. This command is the complement of the **namespace qualifiers** command. It does not check whether the namespace names are, in fact, the names of currently defined namespaces.

namespace upvar *namespace* ?*otherVar myVar* ...?

This command arranges for zero or more local variables in the current procedure to refer to variables in *namespace*. The namespace name is resolved as described in section **NAME RESOLUTION**. The command **namespace upvar $ns a b** has the same behaviour as **upvar 0 ${ns}::a b**, with the sole exception of the resolution rules used for qualified namespace or variable names. **namespace upvar** returns an empty string.

namespace unknown ?*script*?

Sets or returns the unknown command handler for the current namespace. The handler is invoked when a command called from within the namespace cannot be found in the current namespace, the namespace's path nor in the global namespace. The *script* argument, if given, should be a well formed list representing a command name and optional arguments. When the handler is invoked, the full invocation line will be appended to the script and the result evaluated in the context of the namespace. The default handler for all namespaces is **::unknown**. If no argument is given, it returns the handler for the current namespace.

namespace which ?**-command**? ?**-variable**? *name*

Looks up *name* as either a command or variable and returns its fully-qualified name. For example, if *name* does not exist in the current namespace but does exist in the global namespace, this command returns a fully-qualified name in the global namespace. If the command or variable does not exist, this command returns an empty string. If the variable has been created but not defined, such as with the **variable** command or through a **trace** on the variable, this command will return the fully-qualified name of the variable. If no flag is given, *name* is treated as a command name. See the section **NAME RESOLUTION** below for an explanation of the rules regarding name resolution.

WHAT IS A NAMESPACE?

A namespace is a collection of commands and variables. It encapsulates the commands and variables to ensure that they will not interfere with the commands and variables of other namespaces. Tcl has always had one such collection, which we refer to as the *global namespace*. The global namespace holds all global variables and commands. The **namespace eval** command lets you create new namespaces. For example,

```
namespace eval Counter {
    namespace export bump
    variable num 0

    proc bump {} {
        variable num
        incr num
    }
}
```

creates a new namespace containing the variable **num** and the procedure **bump**. The commands and variables in this namespace are separate from other commands and variables in the same program. If there is a command named **bump** in the global namespace, for example, it will be different from the command **bump** in the **Counter** namespace.

Namespace variables resemble global variables in Tcl. They exist outside of the procedures in a namespace but can be accessed in a procedure via the **variable** command, as shown in the example above.

Namespaces are dynamic. You can add and delete commands and variables at any time, so you can build up the contents of a namespace over time using a series of **namespace eval** commands. For example, the following series of commands has the same effect as the namespace definition shown above:

```
namespace eval Counter {
    variable num 0
    proc bump {} {
        variable num
        return [incr num]
    }
}
namespace eval Counter {
    proc test {args} {
        return $args
    }
}
namespace eval Counter {
     rename test ""
}
```

Note that the **test** procedure is added to the **Counter** namespace, and later removed via the **rename** command.

Namespaces can have other namespaces within them, so they nest hierarchically. A nested namespace is encapsulated inside its parent namespace and can not interfere with other namespaces.

QUALIFIED NAMES

Each namespace has a textual name such as **history** or **::safe::interp**. Since namespaces may nest, qualified names are used to refer to commands, variables, and child namespaces contained inside namespaces. Qualified names are similar to the hierarchical path names for Unix files or Tk widgets, except that **::** is used as the separator instead of / or **.**. The topmost or global namespace has the name “” (i.e., an empty string), although **::** is a synonym. As an example, the name **::safe::interp::create** refers to the command **create** in the namespace **interp** that is a child of namespace **::safe**, which in turn is a child of the global namespace, **::**.

If you want to access commands and variables from another namespace, you must use some extra syntax. Names must be qualified by the namespace that contains them. From the global namespace, we might access the **Counter** procedures like this:

```
Counter::bump 5
Counter::Reset
```

We could access the current count like this:

```
puts "count = $Counter::num"
```

When one namespace contains another, you may need more than one qualifier to reach its elements. If we had a namespace **Foo** that contained the namespace **Counter**, you could invoke its **bump** procedure from the global namespace like this:

```
Foo::Counter::bump 3
```

You can also use qualified names when you create and rename commands. For example, you could add a procedure to the **Foo** namespace like this:

```
proc Foo::Test {args} {return $args}
```

And you could move the same procedure to another namespace like this:

```
rename Foo::Test Bar::Test
```

There are a few remaining points about qualified names that we should cover. Namespaces have nonempty names except for the global namespace. **::** is disallowed in simple command, variable, and namespace names except as a namespace separator. Extra colons in any separator part of a qualified name are ignored; i.e. two or more colons are treated as a namespace separator. A trailing **::** in a qualified variable or command name refers to the variable or command named {}. However, a trailing **::** in a qualified namespace name is ignored.

NAME RESOLUTION

In general, all Tcl commands that take variable and command names support qualified names. This means you can give qualified names to such commands as **set**, **proc**, **rename**, and **interp alias**. If you provide a fully-qualified name that starts with a **::**, there is no question about what command, variable, or namespace you mean. However, if the name does not start with a **::** (i.e., is *relative*), Tcl follows basic rules for looking it up:

- **Variable names** are always resolved by looking first in the current namespace, and then in the global namespace.
- **Command names** are always resolved by looking in the current namespace first. If not found there, they are searched for in every namespace on the current namespace's command path (which is empty by default). If not found there, command names are looked up in the global namespace (or, failing that, are processed by the appropriate **namespace unknown** handler.)
- **Namespace names** are always resolved by looking in only the current namespace.

In the following example,

```
set traceLevel 0
namespace eval Debug {
    printTrace $traceLevel
}
```

Tcl looks for **traceLevel** in the namespace **Debug** and then in the global namespace. It looks up the command **printTrace** in the same way. If a variable or command name is not found in either context, the name is undefined. To make this point absolutely clear, consider the following example:

```
set traceLevel 0
namespace eval Foo {
    variable traceLevel 3

    namespace eval Debug {
        printTrace $traceLevel
    }
}
```

Here Tcl looks for **traceLevel** first in the namespace **Foo::Debug**. Since it is not found there, Tcl then looks for it in the global namespace. The variable **Foo::traceLevel** is completely ignored during the name resolution process.

You can use the **namespace which** command to clear up any question about name resolution. For example, the command:

```
namespace eval Foo::Debug {namespace which -variable traceLevel}
```

returns **::traceLevel**. On the other hand, the command,

```
namespace eval Foo {namespace which -variable traceLevel}
```

returns **::Foo::traceLevel**.

As mentioned above, namespace names are looked up differently than the names of variables and commands. Namespace names are always resolved in the current namespace. This means, for example, that a **namespace eval** command that creates a new namespace always creates a child of the current namespace unless the new namespace name begins with **::**.

Tcl has no access control to limit what variables, commands, or namespaces you can reference. If you provide a qualified name that resolves to an element by the name resolution rule above, you can access the element.

You can access a namespace variable from a procedure in the same namespace by using the **variable** command. Much like the **global** command, this creates a local link to the namespace variable. If necessary, it also creates the variable in the current namespace and initializes it. Note that the **global** command only creates links to variables in the global namespace. It is not necessary to use a **variable** command if you always refer to the namespace variable using an appropriate qualified name.

IMPORTING COMMANDS

Namespaces are often used to represent libraries. Some library commands are used so frequently that it is a nuisance to type their qualified names. For example, suppose that all of the commands in a package like BLT are contained in a namespace called **Blt**. Then you might access these commands like this:

```
Blt::graph .g -background red
Blt::table . .g 0,0
```

If you use the **graph** and **table** commands frequently, you may want to access them without the **Blt::** prefix. You can do this by importing the commands into the current namespace, like this:

```
namespace import Blt::*
```

This adds all exported commands from the **Blt** namespace into the current namespace context, so you can write code like this:

```
graph .g -background red
table . .g 0,0
```

The **namespace import** command only imports commands from a namespace that that namespace exported with a **namespace export** command.

Importing *every* command from a namespace is generally a bad idea since you do not know what you will get. It is better to import just the specific commands you need. For example, the command

```
namespace import Blt::graph Blt::table
```

imports only the **graph** and **table** commands into the current context.

If you try to import a command that already exists, you will get an error. This prevents you from importing the same command from two different packages. But from time to time (perhaps when debugging), you may want to get around this restriction. You may want to reissue the **namespace import** command to pick up new commands that have appeared in a namespace. In that case, you can use the **-force** option, and existing commands will be silently overwritten:

```
namespace import -force Blt::graph Blt::table
```

If for some reason, you want to stop using the imported commands, you can remove them with a **namespace forget** command, like this:

```
namespace forget Blt::*
```

This searches the current namespace for any commands imported from **Blt**. If it finds any, it removes them. Otherwise, it does nothing. After this, the **Blt** commands must be accessed with the **Blt::** prefix.

When you delete a command from the exporting namespace like this:

```
rename Blt::graph ""
```

the command is automatically removed from all namespaces that import it.

EXPORTING COMMANDS

You can export commands from a namespace like this:

```
namespace eval Counter {
    namespace export bump reset
    variable Num 0
    variable Max 100

    proc bump {{by 1}} {
        variable Num
        incr Num $by
        Check
        return $Num
    }
    proc reset {} {
        variable Num
        set Num 0
    }
    proc Check {} {
        variable Num
        variable Max
        if {$Num > $Max} {
            error "too high!"
        }
    }
}
```

The procedures **bump** and **reset** are exported, so they are included when you import from the **Counter** namespace, like this:

```
namespace import Counter::*
```

However, the **Check** procedure is not exported, so it is ignored by the import operation.

The **namespace import** command only imports commands that were declared as exported by their namespace. The **namespace export** command specifies what commands may be imported by other namespaces. If a **namespace import** command specifies a command that is not exported, the command is not imported.

SCOPED SCRIPTS

The **namespace code** command is the means by which a script may be packaged for evaluation in a namespace other than the one in which it was created. It is used most often to create event handlers, Tk bindings, and traces for evaluation in the global context. For instance, the following code indicates how to direct a variable trace callback into the current namespace:

```
namespace eval a {
    variable b
    proc theTraceCallback { n1 n2 op } {
        upvar 1 $n1 var
        puts "the value of $n1 has changed to $var"
        return
    }
    trace add variable b write [namespace code theTraceCallback]
}
set a::b c
```

When executed, it prints the message:

```
the value of a::b has changed to c
```

ENSEMBLES

The **namespace ensemble** is used to create and manipulate ensemble commands, which are commands formed by grouping subcommands together. The commands typically come from the current namespace when the ensemble was created, though this is configurable. Note that there may be any number of ensembles associated with any namespace (including none, which is true of all namespaces by default), though all the ensembles associated with a namespace are deleted when that namespace is deleted. The link between an ensemble command and its namespace is maintained however the ensemble is renamed.

Three subcommands of the **namespace ensemble** command are defined:

namespace ensemble create ?*option value ...*?
: Creates a new ensemble command linked to the current namespace, returning the fully qualified name of the command created. The arguments to **namespace ensemble create** allow the configuration of the command as if with the **namespace ensemble configure** command. If not overridden with the **-command** option, this command creates an ensemble with exactly the same name as the linked namespace. See the section **ENSEMBLE OPTIONS** below for a full list of options supported and their effects.

namespace ensemble configure *command* ?*option*? ?*value ...*?
: Retrieves the value of an option associated with the ensemble command named *command*, or updates some options associated with that ensemble command. See the section **ENSEMBLE OPTIONS** below for a full list of options supported and their effects.

namespace ensemble exists *command*
: Returns a boolean value that describes whether the command *command* exists and is an ensemble command. This command only ever returns an error if the number of arguments to the command is wrong.

When called, an ensemble command takes its first argument and looks it up (according to the rules described below) to discover a list of words to replace the ensemble command and subcommand with. The resulting list of words is then evaluated (with no further substitutions) as if that was what was typed originally (i.e. by passing the list of words through **Tcl_EvalObjv**) and returning the result of the command. Note that it is legal to make the target of an ensemble rewrite be another (or even the same) ensemble command. The ensemble command will not be visible through the use of the **uplevel** or **info level** commands.

ENSEMBLE OPTIONS

The following options, supported by the **namespace ensemble create** and **namespace ensemble configure** commands, control how an ensemble command behaves:

-map
: When non-empty, this option supplies a dictionary that provides a mapping from subcommand names to a list of prefix words to substitute in place of the ensemble command and subcommand words (in a manner similar to an alias created with **interp alias**; the words are not reparsed after substitution); if the first word of any target is not fully qualified when set, it is assumed to be relative to the *current* namespace and changed to be exactly that (that is, it is always fully qualified when read). When this option is empty, the mapping will be from the local name of the subcommand to its fully-qualified name. Note that when this option is non-empty and the **-subcommands** option is empty, the ensemble subcommand names will be exactly those words that have mappings in the dictionary.

-parameters
: This option gives a list of named arguments (the names being used during generation of error messages) that are passed by the caller of the ensemble between the name of the ensemble and the subcommand argument. By default, it is the empty list.

-prefixes
: This option (which is enabled by default) controls whether the ensemble command recognizes unambiguous prefixes of its subcommands. When turned off, the ensemble command requires exact matching of subcommand names.

-subcommands
: When non-empty, this option lists exactly what subcommands are in the ensemble. The mapping for each of those commands will be either whatever is defined in the **-map** option, or to the command with the same name in the namespace

linked to the ensemble. If this option is empty, the subcommands of the namespace will either be the keys of the dictionary listed in the **-map** option or the exported commands of the linked namespace at the time of the invocation of the ensemble command.

-unknown

When non-empty, this option provides a partial command (to which all the words that are arguments to the ensemble command, including the fully-qualified name of the ensemble, are appended) to handle the case where an ensemble subcommand is not recognized and would otherwise generate an error. When empty (the default) an error (in the style of Tcl_GetIndexFromObj) is generated whenever the ensemble is unable to determine how to implement a particular subcommand. See UNKNOWN HANDLER BEHAVIOUR for more details.

The following extra option is allowed by **namespace ensemble create**:

-command

This write-only option allows the name of the ensemble created by **namespace ensemble create** to be anything in any existing namespace. The default value for this option is the fully-qualified name of the namespace in which the **namespace ensemble create** command is invoked.

The following extra option is allowed by **namespace ensemble configure**:

-namespace

This read-only option allows the retrieval of the fully-qualified name of the namespace which the ensemble was created within.

UNKNOWN HANDLER BEHAVIOUR

If an unknown handler is specified for an ensemble, that handler is called when the ensemble command would otherwise return an error due to it being unable to decide which subcommand to invoke. The exact conditions under which that occurs are controlled by the **-subcommands**, **-map** and **-prefixes** options as described above.

To execute the unknown handler, the ensemble mechanism takes the specified **-unknown** option and appends each argument of the attempted ensemble command invocation (including the ensemble command itself, expressed as a fully qualified name). It invokes the resulting command in the scope of the attempted call. If the execution of the unknown handler terminates normally, the ensemble engine reparses the subcommand (as described below) and tries to dispatch it again, which is ideal for when the ensemble's configuration has been updated by the unknown subcommand handler. Any other kind of termination of the unknown handler is treated as an error.

The result of the unknown handler is expected to be a list (it is an error if it is not). If the list is an empty list, the ensemble command attempts to look up the original subcommand again and, if it is not found this time, an error will be generated just as if the **-unknown** handler was not there (i.e. for any particular invocation of an ensemble, its unknown handler will be called at most once.) This makes it easy for the unknown handler to update the ensemble or its backing namespace so as to provide an implementation of the desired subcommand and reparse.

When the result is a non-empty list, the words of that list are used to replace the ensemble command and subcommand, just as if they had been looked up in the **-map**. It is up to the unknown handler to supply all namespace qualifiers if the implementing subcommand is not in the namespace of the caller of the ensemble command. Also note that when ensemble commands are chained (e.g. if you make one of the commands that implement an ensemble subcommand into an ensemble, in a manner similar to the text widget's tag and mark subcommands) then the rewrite happens in the context of the caller of the outermost ensemble. That is to say that ensembles do not in themselves place any namespace contexts on the Tcl call stack.

Where an empty **-unknown** handler is given (the default), the ensemble command will generate an error message based on the list of commands that the ensemble has defined (formatted similarly to the error message from Tcl_GetIndexFromObj). This is the error that will be thrown when the subcommand is still not recognized during reparsing. It is also an error for an **-unknown** handler to delete its namespace.

EXAMPLES

Create a namespace containing a variable and an exported command:

```
namespace eval foo {
    variable bar 0
    proc grill {} {
        variable bar
        puts "called [incr bar] times"
    }
    namespace export grill
}
```

Call the command defined in the previous example in various ways.

```
# Direct call
::foo::grill

# Use the command resolution path to find the name
namespace eval boo {
    namespace path ::foo
    grill
}

# Import into current namespace, then call local alias
namespace import foo::grill
grill

# Create two ensembles, one with the default name and one with a
# specified name.  Then call through the ensembles.
namespace eval foo {
    namespace ensemble create
    namespace ensemble create -command ::foobar
}
foo grill
foobar grill
```

Look up where the command imported in the previous example came from:

```
puts "grill came from [namespace origin grill]"
```

Remove all imported commands from the current namespace:

```
namespace forget {*}[namespace import]
```

Create an ensemble for simple working with numbers, using the **-parameters** option to allow the operator to be put between the first and second arguments.

```
namespace eval do {
    namespace export *
    namespace ensemble create -parameters x
    proc plus  {x y} {expr { $x + $y }}
    proc minus {x y} {expr { $x - $y }}
}

# In use, the ensemble works like this:
puts [do 1 plus [do 9 minus 7]]
```

SEE ALSO

interp, **upvar**, **variable**

KEYWORDS

command, ensemble, exported, internal, variable